AD-A037 641   CORNELL UNIV ITHACA N Y DEPT OF COMPUTER SCIENCE   F/G 9/2

AN EVALUATION OF PROCESS AND EXPERIMENT AUTOMATION REALTIME LAN--ETC(U)

JAN 77 J H WILLIAMS

UNCLASSIFIED   NL

1 OF 1

ADA037 641

5,3,15

1

ADA037641

6

An Evaluation of PEARL

Process and Experiment Automation Realtime Language (PEARL),

J.H. Williams

10

Computer Science Department

Cornell University

12 18

January 1977

11

A037 634

This report was prepared for the U.S. Army Electronics Command at Fort Monmouth, New Jersey with the financial support of the Scientific Services Program.

407 072

1. Overview of the language

All of the information about PEARL which is used in this report was obtained from the PEARL Language Description [ESG-76]. Work on PEARL (Process and Experiment Automation Realtime Language) was begun in 1968. The first description of it was published in 1970. Since that time the language has undergone revisions, has been implemented on eight different process control computers, and has been subsetted for specific application to embedded computer systems with realtime constraints. It is this subset language which is described in [ESG-76] and which is discussed in this report.

The language was designed to provide systems engineers with a high level means of specifying programs for embedded applications. For that reason it emphasizes the areas of input-output specification and realtime processing and avoids many of the more complex and powerful constructs and capabilities of high level programming languages. In this latter, algorithmic aspect it has the appearance of a severely restricted dialect of PL/I.

To page 2

It is the conclusion of this report that PEARL is not a viable candidate for the DoD common programming language effort, but that the experience gained from its unique approach to input-output specification and realtime control should prove valuable to the forthcoming design efforts.

In this report, we attempt to point out some of the weaknesses which make the language ill-suited for modification or adoption as the DoD common high level algorithmic language. At the same time we hope to encourage its further study by the eventual language design teams.

2. Language features

Perhaps the most striking feature of PEARL is its method of factoring a program into two pieces, one hardware dependent and one hardware independent. These are called the system division and the problem division respectively. Within the system division the programmer specifies the particular connections between the input-output devices and their data paths. A rather elaborate formalism is developed for making these specifications, including the ability to use mnemonic names for particular bits, bytes, signals, devices, etc. Then in the problem division the programmer can use these mnemonic identifiers to control the input-output activity. This allows an unusual degree of machine independence in the input-output sections of the program; it is conceivable that one could change the hardware configuration and not have to make any modifications at all to the problem division of a running program.

The language of the problem division of a PEARL program is essentially a PL/I derivative. We describe its features in the remainder of this section, in particular its data-types, data structures, control structures, declarations, program structure, and some of its unique features.

a) Data Types

There are six basic data types with the normal accompanying arithmetic and relational operators.

i)	FIXED	for integer numbers
ii)	FLOAT	for floating point numbers
iii)	BIT(n)	for bit strings
iv)	CHAR(n)	for character strings
v)	CLOCK	for time of day
vi)	DUR	for interval of time

It is possible to denote constants of any of these types, 14:21:06 and 2 HRS 31 MIN being denotations of CLOCK and DUR constants respectively.

In addition, there are five other data types associated with input-output and task control. They differ from the above six types in that there are no operators in the language which manipulate these types. They are:

DEVICE-MODE

FILE-MODE

INTERRUPT-MODE

SIGNAL-MODE

SEMA

More will be said about these types in the discussion on declarations and program structure.

b) Data Structures

It is possible to form collections of the basic data types using either arrays or structures, the former being homogeneous. The array mechanism is essentially that of PL/I with the important difference that all array bounds must be constants. This is true of formal parameter specifications as well, seriously limiting the usefulness of procedures with array parameters. There are no provisions for manipulating sub-arrays or slices. Except for input-output, there are no operators which manipulate entire arrays.

The structure mechanism allows the programmer to create collections of non-homogeneous simple types.

DCL ST STRUCT (N1 FLOAT,N2 BIT(3)) creates a structure called ST which consists of a floating part N1 and a 3-bit string N2 . Components of structures cannot themselves be arrays or structures, nor can a structure have a varying composition as with PASCAL records.

There is no mode declaration facility in PEARL. Basically each STRUCT definition creates a new "mode". However it is possible

to use the LIKE attribute as in PL/I to overcome some of the inconvenience caused by the lack of explicit mode declarations.

c) Control structures

These are essentially PL/I with some restrictions and the addition of a case statement.

i) Goto-statement

It is not permitted to use a GOTO statement to exit a task or procedure.

ii) Conditional-statement

This is the usual IF statement with an optional ELSE and a required delimiter FIN. The conditional expression is of type BIT(1) with '1'B representing true as in PL/I.

iii) Case-statement

This is a generalization of the IF statement wherein the conditional expression is of type integer, and the appropriate statement is executed. There is no provision for case statements with conditional expressions of other data types.

iv) Iteration-statement

This statement is of the form:

```
FOR i FROM a BY b TO c WHILE d
   REPEAT
         stmts
   END;
```

v) Procedures and tasks.

The procedure mechanism is similar to PL/I but with some rather severe restrictions. Parameters can be passed by value or by reference. Neither procedures nor labels may be passed as parameters. The type of the actual parameter must match that of the formal parameter exactly. Thus, procedures cannot be written which will work for arrays of differing sizes. These restrictions imply the need for rather clumsy and inefficient error handling mechanisms in programs. Since procedures can neither exit to indicate abnormal conditions nor call error handling procedures passed to them, some global or parametric flags will need to be raised and interrogated to accomplish error handling. This will lead to run time inefficiences.

The task mechanism is very much simpler than that of PL/I, due to the restrictions on program structure which are discussed below. What remains when one removes the complications of dynamic environments and requires all tasks and procedures to be declared at level zero is a rudimentary but effective mechanism for describing actions associated with real time conditions in the embedded computer system environment. It is possible to schedule tasks AT real times, AFTER particular events or durations of time, WHEN external interrupts occur, ALL some duration UNTIL some clock time, etc. This capability is, of course, greatly enhanced by the basic types, CLOCK and DUR and the ability to manipulate them. Tasks can be suspended, reactivated, and terminated and can be synchronized via the semaphore data types as in many languages.

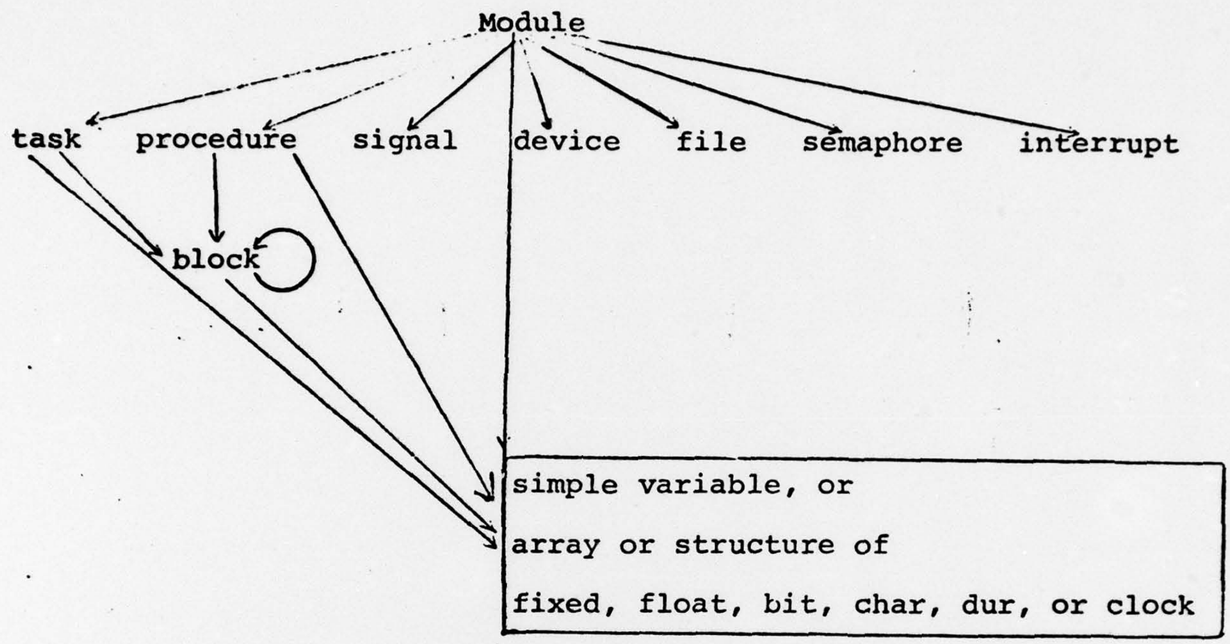
d) Declarations

The declaration mechanism is that of PL/I with some changes. It suffers the same non-orthogonality conditions as does PL/I. In addition, variable names can be of any length, but only the first six characters have any significance. This is an unnecessary concession to implementation shortcuts which can lead to disastrous

consequences in program clarity. There is a modifier to indicate constant declarations. (Curiously a "variable" can be declared INV or invariable!) There is also a mechanism for specifying the attributes of a variable which has been declared in some other program module and is to be considered "common" to this module. In addition, variables can be initialized with the INIT attribute. This initial value is used every time the block is re-entered.

e) Program structure

PEARL programs consist of modules which are separate entities connected only through the common (or SPC for specification) mechanism of their declarations. They can thus be separately compiled. Each module consists of a problem division and a system division as discussed earlier. A module is made up of declarations of variables, tasks and procedures. Tasks and procedures consist of declarations of restricted types of variables and blocks of statements. The only syntactically self-nesting program construct is the block. We illustrate this pictorially in the following diagram. The arrow relationship is read "can contain an instance of."



The most significant restriction of this program structure is the lack of nested procedure definitions.

f) Miscellaneous language features

A variable may be declared "global" and shared with another module. However, any variable so declared must be declared at the module level. Thus if one procedure (or task or nested block within a module) is to be able to share a declaration with some

other module, <u>all</u> tasks, procedures and blocks in that module will also have access to the shared object.

It is possible to specify the length of integer and floating variables. This is done at the module level.

It is not permitted to perform the usual mixed mode arithmetic operations. For example, if A is FLOAT then A+1 is illegal. It is possible to do a few operations of mixed mode, for example, DUR+CLOCK→CLOCK. There are a number of explicit conversion routines, <u>eg</u> A+FLOAT(1) would be legal.

It is possible to declare a procedure to be reentrant "in case several tasks wish to use the procedure simultaneously." The manual gives no other hint of whether procedures can be recursive. This question will be considered more completely in the next section.

3. Language characteristics

a) Machine independence

The language has been designed to be very much machine independent. This is accomplished in large part by the separate system division of program modules. There are other design features which contribute to this independence, such as the provision for declaring the lengths of integers and floating point numbers rather than a mechanism such as DOUBLE for extended precision which is inherently machine dependent. Unfortunately, there are some small machine dependencies built into the language; eg the implicit conversion which takes place during an assignment of the form, FIXED←FLOAT, depends on the wordsize of the machine for its definition. All in all however, the problem division of a PEARL module is highly machine independent, which is rather remarkable for a language designed specifically for realtime control of embedded computer systems.

b) Separate compilation

This was evidently one of the design criteria of PEARL. Because of its simple static storage made possible by the program structure and its independent modules, it is completely separately compilable with almost no work left for the loader.

c) Language efficiency and size

Again, because of the program structure, the limited type-declaration facility, and the compile time typing of all variables, PEARL compilers should produce very efficient object code. As mentioned earlier, the Spartan nature of the language could lead to a somewhat inefficient expression of algorithms at the source level. The language is small enough to allow a compiler to operate efficiently on a small machine. Ease of implementing seems to have been a goal of the language design, sometimes leading to extremes as in the case of variable identifier lengths.

4. The impact of PEARL on the DoD common language effort

PEARL does not come close to satisfying the many requirements of the Tinman document [Fi-76]. It would take a great amount of modification and enhancement to make it a viable candidate for the DoD's common language. Indeed, it is essentially a derivative of PL/I with many of the more powerful (and sophisticated) features removed. As such it is not nearly as rich or modern a basic language as, for example, Pascal or Algol-68.

However, the PEARL language experience should have a decided impact on the DoD effort. Since the language was designed specifically for embedded computer systems, the experience gained from its use, particularly the use of its system division, should be valuable to any team attempting to adapt an existing language to meet the Tinman requirements. This language is unique in its approach to specifying input-output hardware in a high order language. This is not to suggest that the actual syntax of PEARL is the best that can be done and should be adopted by the forthcoming design efforts. In fact, at the recent language workshop held at Cornell University, Professor Eberhard Wegner indicated that work is currently underway to radically revise the input-output handling and specification mechanisms of PEARL [Weg-76]. However, the

experience gained from the use of this language and the reasons for the current revisions should prove to be useful input to those design efforts.

5. References

[ESG-76]

ESG Elektronik-System-Gesellschaft mb H, PEARL Subset for Avionic Applications, Munich, Germany, June 1976.

[Fis-76]

Fisher, D.A., "A common programming language for the Department of Defense - background and technical requirements", IDA report number P-1191, June 1976.

[Weg-76]

Wegner, E., *comments made at the Department of Defense* Language Workshop, Cornell University, October 1976.